US Army Corps
of Engineers
Construction Engineering
Research Laboratories

USACERL Technical Report FF-95/05
January 1995

Fundamental Database Process

Issues in Object-Oriented Knowledge Representation

by
R. Alan Whitehurst
Mehdi T. Harandi
Jane Wang

The Department of Defense has identified simulation and modeling as technologies critical to national security. The U.S. Army Construction Engineering Research Laboratories (USACERL) has recognized the need to provide a more integrated approach to model development and to extend the object-oriented representation to allow more sophisticated uses of knowledge-based tools that support the complex requirements of simulation and modeling.

This research investigated fundamental issues in knowledge representation and object-oriented modeling to provide an extension of the object-oriented formalism that would better support knowledge-based programming tools and techniques. Object-oriented and frame-based approaches are compared. Three aspects of these approaches are examined: the philosophy behind the approach, its methodology, and its implementation. An augmented object-oriented representation is proposed to provide a greater capability to express knowledge about objects, to provide better structure to organize knowledge, and to allow tools to be built to reason about the knowledge stored in the representation.

USER EVALUATION OF REPORT

REFERENCE: USACERL Technical Report FF-95/05, *Fundamental Database Process: Issues in Object-Oriented Knowledge Representation*

Please take a few minutes to answer the questions below, tear out this sheet, and return it to USACERL. As user of this report, your customer comments will provide USACERL with information essential for improving future reports.

1. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which report will be used.)

____

____

____

2. How, specifically, is the report being used? (Information source, design data or procedure, management procedure, source of ideas, etc.)

____

____

3. Has the information in this report led to any quantitative savings as far as manhours/contract dollars saved, operating costs avoided, efficiencies achieved, etc.? If so, please elaborate.

____

____

4. What is your evaluation of this report in the following areas?

a. Presentation: ______________________________

b. Completeness: ______________________________

c. Easy to Understand: ______________________________

d. Easy to Implement: ______________________________

e. Adequate Reference Material: ______________________________

f. Relates to Area of Interest: ______________________________

g. Did the report meet your expectations? ______________________________

h. Does the report raise unanswered questions? ______________________________

i. General Comments. (Indicate what you think should be changed to make this report and future reports of this type more responsive to your needs, more usable, improve readability, etc.)

____

____

____

____

____

____

5. If you would like to be contacted by the personnel who prepared this report to raise specific questions or discuss the topic, please fill in the following information.

Name: ______________________________

Telephone Number: ______________________________

Organization Address: ______________________________

____

____

6. Please mail the completed form to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORIES
ATTN: CECER-IMT
P.O. Box 9005
Champaign, IL 61826-9005

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE January 1995	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE
Fundamental Database Process: Issues in Object-Oriented Knowledge Representation

5. FUNDING NUMBERS
4A161102
AT23
SE-EB2

6. AUTHOR(S)
R. Alan Whitehurst, Mehdi T. Harandi, and Jane Wang

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Construction Engineering Research Laboratories (USACERL)
P.O. Box 9005
Champaign, IL 61826-9005

8. PERFORMING ORGANIZATION REPORT NUMBER
FF-95/05

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
USACERL
ATTN: CECER-ECS
PO Box 9005
Champaign, IL 61826-9005

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Copies are available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Department of Defense has identified simulation and modeling as technologies critical to national security. The U.S. Army Construction Engineering Research Laboratories (USACERL) has recognized the need to provide a more integrated approach to model development and to extend the object-oriented representation to allow more sophisticated uses of knowledge-based tools that support the complex requirements of simulation and modeling.

This research investigated fundamental issues in knowledge representation and object-oriented modeling to provide an extension of the object-oriented formalism that would better support knowledge-based programming tools and techniques. Object-oriented and the frame-based approaches are compared. Three aspects of these approaches are examined: the philosophy behind the approach, its methodology, and its implementation. An augmented object-oriented representation is proposed to provide a greater capability to express knowledge about objects, to provide better structure to organize knowledge, and to allow tools to be built to reason about the knowledge stored in the representation.

14. SUBJECT TERMS
object-oriented modeling
simulation
knowledge-based systems

15. NUMBER OF PAGES
54

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

Foreword

This study was conducted under Project 4A162720AT23, "Basic Research in Military Construction"; Work Unit SE-EB2, "Advanced Collaborative Systems."

The work was performed by the Facility Management Division (FF) of the Infrastructure Laboratory (FL), U.S. Army Construction Engineering Research Laboratories (USACERL). Dr. Mehdi T. Harandi and Jane Wang are associated with the Department of Computer Science, College of Engineering, University of Illinois, Urbana. Alan Moore is Chief, CECER-FF, and Acting Chief, CECER-FL. The USACERL technical editor was William J. Wolfe, Information Management Office.

LTC David J. Rehbein is Commander and Acting Director of USACERL, and Dr. Michael J. O'Connor is Technical Director.

Contents

1 Introduction

1.1 Background

The Department of Defense has identified simulation and modeling as technologies critical to national security [22]. Research in this area by the U.S. Army Construction Engineering Research Laboratories (USACERL) has included the development of combat engineering modeling and simulation technologies in support of the U.S. Army Engineer School (USAES), and of simulation language technologies in support of the U.S. Army Training and Doctrine Command (TRADOC) [14, 8, 7, 12, 11, 13, 9, 10, 16, 26]. During the course of this research, USACERL recognized the need to provide a more integrated approach to model development and to extend the object-oriented representation to allow more sophisticated uses of knowledge-based tools to support the complex requirements of simulation and modeling.

To address the need for an integrated approach to simulation technologies, USACERL is developing an Integrated Systems Language Environment (ISLE), a software engineering environment based upon an object-oriented database. ISLE integrates a number of software technologies into a single environment that will meet the requirements for the development of the next-generation of integrated modeling and simulation applications. These technologies include: object-oriented programming, process-based discrete event simulation, and knowledge-based programming.

One goal in creating ISLE is to develop an environment that is accessible to groups of cooperating domain experts who are not necessarily computer programmers. To meet this goal, a suite of knowledge-based tools are envisioned that will help users navigate class hierarchies, thereby extending, or *specializing*, the existing software artifacts to meet specific requirements. For the knowledge-based tools envisioned for the ISLE environment to reason about the models and simulations created under ISLE, a single data/knowledge representation is sought that would support both the object-oriented notion of encapsulation and inheritance, and the knowledge-based paradigm of declarative programming and expressibility.

1.2 Objective

The objective of this research was to investigate fundamental issues in knowledge representation and object-oriented modeling to provide an extension of the object-oriented formalism that would better support knowledge-based programming tools and techniques, specifically, in the ISLE software engineering environment.

1.3 Approach

Object-oriented and frame-based knowledge representation schemes were compared (Chapter 2). The facilities of knowledge representation languages, such as frames and semantic networks, were analyzed and the representations used by object-oriented programming languages were extended to provide better support for these knowledge-based capabilities. The syntax and semantics of an extension of object-oriented representation, which incorporates many features of the frame-based approach, were proposed (Chapter 3). The added capabilities include the ability to:

- attach more semantic information to the structure/operations of a class specification
- combine class specifications with more flexibility
- maintain information to allow the navigation of the inheritance network and other relationships between objects to facilitate reasoning.

Directions for this and future research were outlined (Chapter 4).

1.4 Mode of Technology Transfer

It is anticipated that the concepts developed under this extended object-oriented knowledge representation will be incorporated into the IMPORT/DOME languages and the ISLE software engineering environment.

2 Knowledge Representations

The representational framework in which a problem is phrased has a definite impact on the approach taken, and consequently, on the ease of solution [27]. This study seeks a knowledge-representation formalism that will be both powerful and flexible enough to capture the knowledge in the domain of simulation and modeling, and that will still mesh well with the object-oriented programming paradigm.

A programming paradigm is considered to be object-oriented if it exhibits three characteristics:

- encapsulation
- information hiding
- polymorphism.

Object-oriented representations are encapsulated, in that the data and the operations that manipulate that data are contained in a single syntactic specification (usually refered to as the *class* of the object) and access to the data can occur only through well-defined interfaces. Information hiding refers to the ability to make public a particular interface, and to hide the details of the implementation. Finally, polymorphism is the ability to allow a particular object instance to react to a request for processing as is most appropriate to the object's class. These three attributes combine to support and encourage *software reuse*, which is one of the main contributions of the object-oriented approach.

Object classes occupy a position in a class hierarchy (in the case of *single inheritance*) or directed acyclic graph (in the case of *multiple inheritance*) that relates classes based on common features. Therefore, an object is a set of data types and operations that is related to other objects through inheritance. We are searching for knowledge-representation formalism that will allow us to retain the benefits of the object-oriented paradigm, while supporting extended semantic reasoning about objects and relations.

At first glance, a frame-based representation seems a natural choice for integration with an object-oriented language; frames are structured much like objects and incorporate a version of inheritance. One of the problems encountered in attempting a thorough comparison of the two approaches is that there seems to be multiple definitions of just what "frame" and "object-oriented" are supposed to mean. In [6], Hayes comments on the confusion this causes with respect to frames when he writes:

> ... It is not at all clear now what frames are, or were ever intended to be. I will assume that frames were put forward as a formal language for expressing knowledge ... but it is important to distinguish this from two other possible interpretations ... which one might call the metaphysical and the heuristic.

Hayes goes on to describe these three uses of the term "frame." According to Hayes, Metaphysical frames embody a philosophy that influences how knowledge is structured and what knowledge is determined to be of importance. Formal frames are a notation for expressing knowledge, and Heuristic frames are a computational device for organizing stored representations and managing the process of retrieval and inference.

It is interesting to note that a confusion regarding terminology is not limited to the concept of frames:

> The explosion of interest in object-oriented approaches in the last few years has led to a proliferation of definitions and interpretations of this much-used and much-abused term. As a consequence, it can be very difficult for a newcomer to understand and evaluate what is meant by a claim that a programming language or a piece of software or a user interface is 'object-oriented.' [21]

Johnson [15] describes three separate uses of the term "object-oriented," which he classifies as the mystical, European, and American views. In the mystical view, "object-oriented" refers to a philosophy of computing in which systems are composed entirely of objects that can communicate only by sending messages. In the European view, "object-oriented" is a *modeling methodology* where each entity in a system being modeled is represented by an entity in the program. Finally, the American view defines "object-oriented" as programming languages or systems that exhibit certain characteristics and promote the use of a certain set of programming *techniques*, such as: data abstraction, encapsulation, polymorphism, and inheritance.

Figure 2.1 shows the relationship between the two sets of definitions. These definitions for both frame and object-oriented approaches can be arranged in order of abstraction (metaphysical and mystical being the most abstract). At the highest level of abstraction, both concepts involve a philosophy of knowledge selection and encapsulation. At the intermediate level, the definitions involve a methodology for knowledge expression. At the lowest level, both definitions involve a set of strategies or techniques of reasoning or computation. Unfortunately, this mapping is not

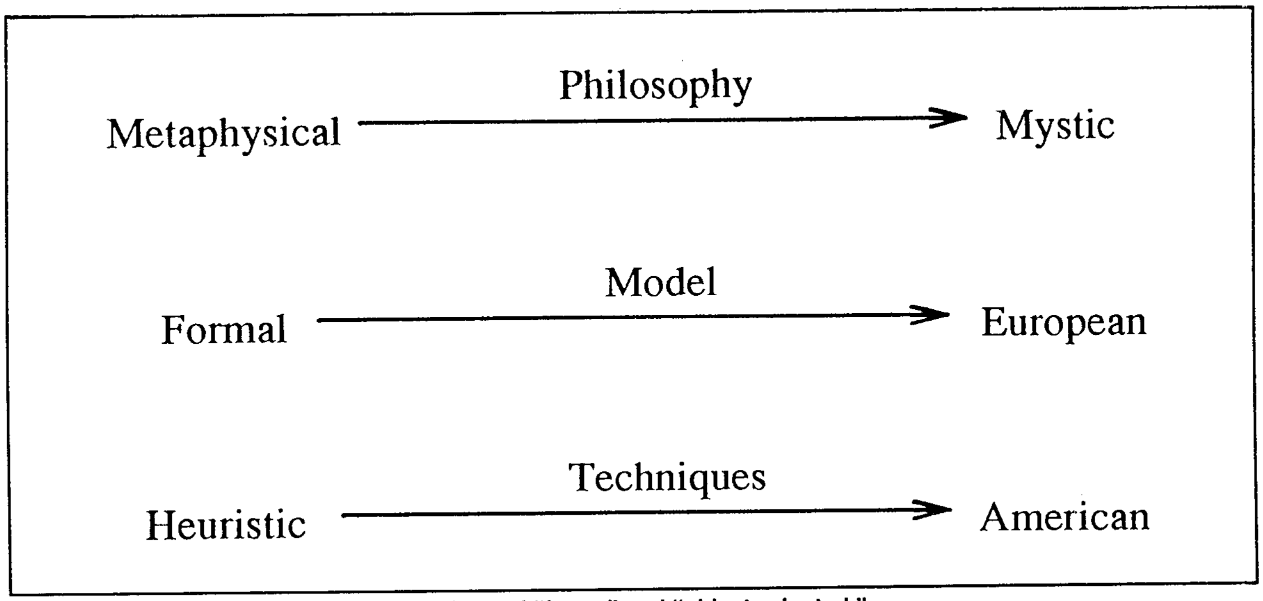

Figure 2.1. Relationship between definitions of "frame" and "object-oriented."

straightforward. Certain aspects of the mystical view of objects are more closely related to expressibility and language constructs, while some aspects of the European view (of objects as a modeling methodology) have more in common with the philosophy of frames. The remainder of this report will attempt to adopt Hayes' framework and address the comparison between objects and frames at each of these three conceptual levels. Although this may be an artificial distinction in some respects, it does provide a framework for structuring a discussion about frame and object concepts.

2.1 Philosophy of Frames and Objects

The original concept of frames in relation to knowledge representation is attributed to Marvin Minsky [19], who felt that logic-oriented approaches, based on collections of simple fragments, were too localized and unstructured to account for common-sense reasoning. Minsky felt that knowledge was inherently interrelated and structured. The essence of Minsky's theory was that when one encounters a new situation, one selects from memory a structure (which Minsky called a frame) that carries with it certain assumptions that are subsequently adapted to fit reality.

According to Minsky, each frame carried information about how to use the frame, what to expect next, and what to do if expectations were not confirmed. Each frame also supplied a set of default values that could be altered to make the prototypical frame correspond to a specific concept or object. Further, these values had associated constraints that affected the way they could be changed and that could propagate the change to other frames. Therefore, knowledge was bundled into a set of interrelated structures, each encapsulating a certain aspect of the knowledge domain. Frames are related into networks of interdependencies, such that changes to the values of one frame may propagate throughout the system. Computation is achieved by altering the values of a frame and allowing those changes to propagate throughout the frame network.

The metaphysical view of object-oriented systems shares much in common with the philosophy of frames as espoused by Minsky. In the object-oriented view, the functionality of a system is realized by a community of cooperative agents, each performing specific functions and communicating with other agents via messages. New behaviors are recognized and accommodated by reproducing existing similar objects and specializing their functionality. Computation is achieved by introducing a message to a particular object, which responds by performing its function and/or passing a set of messages to other objects. At the philosophical level, the fundamental principle behind both frames and objects is the same: knowledge and/or complex behavior can be encoded as a set of simpler, interrelated structures that encapsulate various aspects of the system.

2.2 Modeling Knowledge

A frame is a structure intended to represent a prototypical situation. It is comprised of named slots (or fields), which may themselves contain frames, or which may contain simple data. Through semantics associated with the slot names (e.g., ISA to denote that one thing is an instance of another, like a "dog" is an instance of a "mammal"), a form of inheritance is achieved that enables default reasoning and facilitates the sharing of knowledge. Winston [27] proposed a simple way to conceptualize frame-based representations: "A *frame* is a collection of semantic net nodes and slots that together describe a stereotyped object, act, or event."

Winston's definition implies two important aspects of frame-based representations: first, that they are related to semantic nets; and second, that the notion of inheriting values from a stereotypical prototype is integral to the frame-based approach.

Figure 2.2 illustrates Winston's view of the relationship between semantic networks and frame representations: that frames can be thought of as imposing a structure on top of a semantic network.

This example shows a portion of a semantic network representing knowledge about the relationships between bricks and toy blocks. The ellipses represent objects, and the

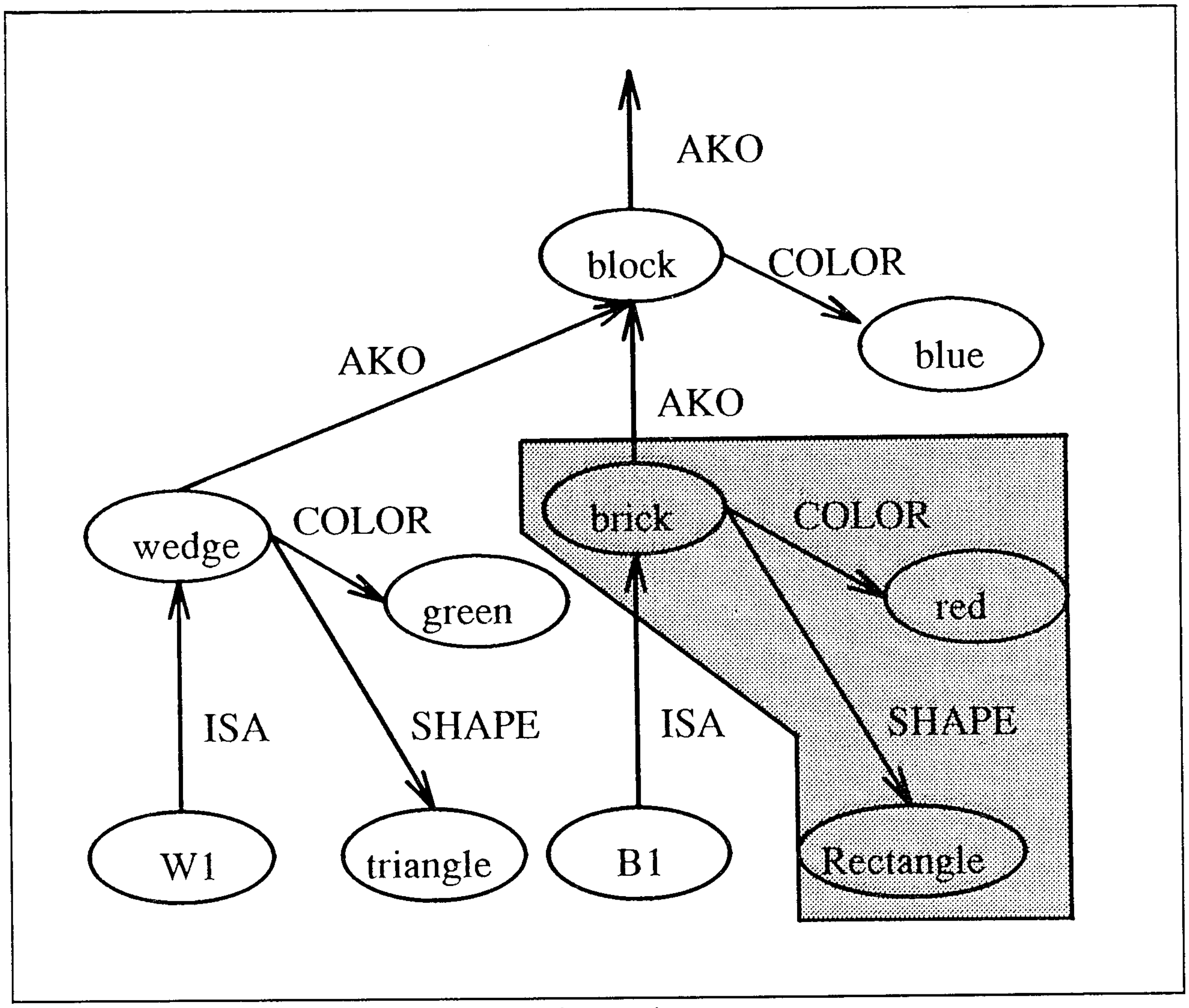

Figure 2.2. Relationship between frames and semantic networks.

arcs represent the relationships between objects; therefore, reading from the semantic network, one sees that a brick is considered to be a kind of block, that bricks are normally red, but that blocks are normally blue, and that that there are kinds of blocks that are not bricks (namely, wedges). The area enclosed by the shaded polygon represents a "brick" frame.

An object is an encapsulation of a certain aspect of the system being modeled, but the emphasis is on the objects in the system rather than the processes they perform. An object is described in terms of its attributes and its behaviors. According to Meyer:

> Object-oriented design may be defined as a technique that, unlike classical design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs. [18]

The conjunction of the attributes of all the objects in a system describes the state of the system. The behavior of an object describes the ways in which that object can affect the state of the system. The functionality of the system is realized by cooperation between these objects. The power of this philosophy is realized in the ability to create a model of reality that preserves a one-to-one mapping between entities in the real-world and entities in the model.

2.3 Implementation

The difficulty in attempting to compare implementations of frame and object systems is that there are so many different frame and object systems, each with its own set of implementation decisions. It is feasible in the scope of this report to describe a frame or object system implementation only in the most general terms.

Frame systems provide a mechanism for structuring knowledge into frames, and organizing frames into hierarchies of prototypical knowledge. Each frame consists of a number of slots, and each slot contains information specific to the frame. A slot may also have an attached predicate that encodes procedural knowledge associated with the frame. However, to realize his concept of frame-based reasoning, Minsky proposed a computational approach that combined: prototypes, defaults, multiple perspectives, analogies, and partial matching [20]. Minsky's ideas lead to the development of a number of knowledge-representation architectures, one of which was the Knowledge Representation Language (KRL) [1]. KRL was built as part of a long-term project whose goals centered around language understanding. The viewpoint of the KRL project was that reasoning is dominated by a process of recognition in which new objects and events are compared to a stored set of expected prototypes, and that the key part of the recognition process was a description matcher that served as a framework for comparing descriptions.

Object-oriented languages provide a set of mechanisms for realizing an object-oriented design and enforcing an object-oriented methodology during implementation. These include:

- encapsulation
- information hiding
- inheritance
- abstraction.

Encapsulation, as has already been discussed, is the ability to structure state and behavior into individual syntactic and semantic packages. Information hiding relates to the restriction that objects only communicate through methods; therefore, an object does not have access to the structure of any other object. If information is needed about another object, the object must be queried through one of its methods. This decouples the internal structure of the object from its external behavior. Inheritance is the ability of one class of objects to be based on another class—in other words, to inherit the structure and functionality of the other class. Finally, abstraction is the ability to create objects that capture abstract aspects of a set of concrete object classes, and to use that abstract object to share its structure and behavior among those similar concrete object classes.

The motivation for incorporating these techniques into software methodology is to improve the software engineering process. This improvement is to be realized by maximizing the ability to reuse software design and implementation.

> ... if one accepts that reusability is essential to better software quality, the object-oriented approach—defined as the construction of software systems as structured collections of abstract data type implementations—provides a promising set of solutions.[18]

Although the motivation behind the concepts is quite different, there are a number of striking similarities between the two approaches:

- Both decompose a system into a collection of objects and relationships.
- Both encapsulate or bundle knowledge into groups of attributes.
- Both use a similar structure (named slot-value pairs).
- Both incorporate a notion of inheritance.
- Both include a notion of individual active entities.

The concept of matching frames to find prototypical ancestors has no analog in object-oriented approaches, and embodies the major difference between the two approaches. A frame-based system uses introspection and default reasoning to dynamically increase and refine its model. When a frame-based system is given a task that it does not expect, ideally it can apply partial matching and analogy to attempt understanding. There is an implicit assumption in the object-oriented approach that the model is consistent and complete.

It is considered an error for an object to receive a request (i.e., a message) that it (or one of its direct ancestors) does not understand. These are design decisions; there is nothing intrinsic in the object model that dictates that unexpected messages must be treated as errors; however, the implications in terms of software reuse refer back to the original motivation behind the development of the two paradigms.

In object-oriented approaches, inheritance and polymorphism provide mechanisms for sharing implementations, with the ultimate goal of producing reusable module, sometimes referred to as "software ICs." In frame-based approaches, inheritance provides a mechanism for prototypical reasoning, where assumptions are inherited and refined as specific information becomes available.

2.4 Summary

This chapter has explored the relationship between knowledge representation techniques based on semantic networks and frames, and the object-oriented software engineering model. This relationship has been examined at three separate levels of abstraction: at the philosophical level, the modeling level, and the implementation level. It was found that the philosophy and modeling aspects of these two formalisms are remarkably similar. However, it is noted that the motivation for the two formalisms are quite different. Frame-based systems are motivated by a desire to model prototypical reasoning, while object-oriented systems desire to enhance the software-engineering process and maximize software reuse. This difference in motivation manifests itself in implementations based on these two systems. In the next chapter, an extension of the object-oriented representation is proposed that integrates many of the facilities commonly found in frame-based implementation systems.

3 Extended Object Representation

3.1 Introduction

The basic framework of the proposed knowledge representation is an augmented object-oriented representation. It is an attempt to provide greater capability to express knowledge about objects, to provide better structure to organize knowledge, and to allow tools to be built to reason about the knowledge stored in the representation. The added features are:

- attaching slot facets to attributes to provide more semantic context
- using perspectives to group together the relevent information about an object
- adding a bidirectional link for the superclass/subclass relation
- adding a bidirectional link for the class/perspective relation
- adding constraint, invariants, security, and integrity checking either on the operation/attribute or on the class itself
- adding bidirectional association and aggregation links to describe the relation between objects.

In the following subsections, each topic is examined individually.

3.2 Increasing Semantic Information

3.2.1 Slot Facets

In comparing frame-based systems with semantic nets [24, 27, 23, 4, 3, 2, 28], one finds that it is possible to attach slot facets to the attributes to provide more semantic content to attributes and classes. Slot facets are used in frame-based systems to let users specify more information about an attribute in addition to its data type. Possible slot facets are cardinality facet, range facet, minimum value facet, maximum value facet, default value facet, prefer value facet, if_needed condition facet, and pre-/post-condition facet [5].

Range facet is used to specify the valid range for the attribute value. Minimum/maximum value facet is used to specify the possible smallest/largest value for the attribute value. Default value facet is used to specify the default value for an attribute when there is no value given for it. When an attribute is accessed and no value is found, if_needed condition facet can be used to specify the way to find the value. If_needed condition facet can also be used to specify security/integrity checking of the attribute value during an updating request. Similarly, the pre/post-condition facet can be used when the attribute is accessed or updated.

In addition to the slot facets similar to what frame-based systems have provided, we also introduce ADD and DELETE facets. They are used to modify the data type of an inherited attribute during specialization of a class or perspective.

3.2.2 Perspectives

Observation shows that, depending on the context in which an object is used, attributes with the same syntax may have different semantic meanings. Also, depending on the context in which an object is used, the object might use a different set of attributes and operations. This implies that attributes and operations of an object can be partitioned into different groups depending on the context in which they are used. This introduces the notion of perspective into the object structure (similar to Winston's work) [27].

Structurally, a perspective is similar to a class. It has its own hierarchy, attributes, and operations. It decides its own subperspectives; and a perspective has its own inheritance. Just like classes, perspectives can also have abstract operations implemented at the object level.

Perspectives may be seen as a way to put all the relevent information about an object together in the object level and then partition this information into different groups according to the context in which they are used. Common attributes for all contexts are left alone and kept with the object as the basic/default component of the object. For example, instead of having a Queue class with three subclasses FIFO_Queue (First-In, First-Out), LIFO_Queue (Last-In, First-Out), and Priority_Queue, a Queue class could have all the attributes and operations common to all queue objects defined in the default component section. Thus three perspectives (FIFO_Queue_P, LIFO_Queue_P, and Priority_Queue_P) will be defined in the Queue class and have the perspective-dependent operations like "enqueue" and "dequeue" defined within each perspective. So if an object is a Queue class with FIFO_Queue_P perspective, then it has the default attributes of the Queue class, and it always adds new elements at the end of the queue and removes elements from the beginning of the queue.

To provide reusability as well as privacy of perspectives, perspectives may be defined either globally or locally. Globally defined perspectives can be shared by different classes and perspectives. Locally defined perspectives are defined within a class definition, and are only visible to the class itself. A global perspective can be used either by being attached to a class definition or by specializing it into subperspectives. Local perspectives can be used when specializing/instantiating the class that defines it. The following section gives an example of how to define and use perspectives.

Constraints among perspectives can be specified using XOR and SET. For instance, FIFO_Queue_P, LIFO_Queue_P, and Priority_Queue_P, in the Queue class, should be mutually exclusive. So, in the definition, we can do "XOR(FIFO_Queue_P ... , LIFO_Queue_P ... , Priority_Queue_P ...)" to specify the mutual exclusive constraints among the perspectives. SETs are used when a class can be instantiated/specialized with any combination of perspectives specified within the set. This provides some multiple perspective inheritance. For example, instead of defining customer class, traveller class, and travelling customer class, a Person class with name, age, and homeAddress (i.e., the common attributes of all three classes) can be defined as the default attributes. And within Person class, Traveller_P perspective and Customer_P perspective, which only contain information relevent to traveller and customer, respectively, can also be defined. While defining these two perspectives in Person class, doing "SET(Traveller_P ... , Customer_P ...)" allows a class to be a subclass of Person class with only Traveller_P perspective, or with only Customer_P perspectives, or with both Traveller_P and Customer_P perspectives, which corresponds to traveller class, customer class, and travelling customer class. A discussion of using XOR and SET is in the inheritance section (p 21).

3.2.3 Bidirectional links for superclass/subclass and class/perspective relation

In the traditional object-oriented system, the only system provided relation is the is_a (superclass/subclass) relation. Subclass knows its superclass(es) and can access information about its superclass(es). But, a class itself has no knowledge of its subclasses, if it has any subclasses at all. From the standpoint of knowledge representation in a class hierarchy, to know more about a class besides the attributes and operations defined within the class itself requires going upward as well as going downward in the tree from that class node.

Going upward in the tree yields information about superclasses (ancestors) that will give more understanding about the current class. For example, the class "American_Car" might only have attributes about the manufacturers, with slot facets listing some constraints about the attributes to ensure that the manufacturers are American companies. Also, the class might have some attributes relating to regulation in the

United States. But these attribute definitions alone do not apply exclusively to a class of cars. Traversing upward in the class hierarchy will reveal that the current node is a subclass of a Car class, with a motor, wheels, doors, speed, ... attributes that suggest the class of cars.

Going downward in the tree from a class node reveals information about subclasses that tell more about the current class. For example, the class "transportation" may have attributes of starting place, destination, and speed. But these attributes may apply to many sub-classes: airplane, automobile, bicycle, and onfoot. (Under an automobile class, information about different individual cars which might reveal that it is a class of cars. So this information can be propagated back up to the transportation class.) Similarly the information in the airplane, bicycle, and other subclasses can be propagated back up to the transportaion class.

With this information, the transportation class is shown to be composed of things that can move objects from one place to another. Thus if the current class is too specific, traversing upward in the hierarchy gives more general information about a class. If the class is too general, then traversing downward reveals more detailed information about the class. In this way, adding bidirectional links for superclass/subclass and class/perspective relation gives a greater capability to reason about classes and perspectives.

3.2.4 Constraints and Invariants

In addition to attributes and operations, there are things about an object that remain true. There is a need somehow to associate invariants with the object representation and to store these properties in the object level. For example, suppose that the data object needs to be sorted at all times. It would be best to store this information in the object level so that it can be used in the programming process.

In traditional object-oriented systems, information about constraints of a class is normally embedded in the operations of the class. Here it is proposed to add constraints, invariants, security, and integrity checking either on the operation/attribute or on the class itself. Constraints specified on the object can be checked by an operator when it is invoked to determine whether the operation should be executed.

3.2.5 Association and Aggregation

Often the pointer to an object is stored in another object's attribute with a descriptive name to describe the relation between two objects [25]. But most of the time what is really meant is that there is an association relation between these two objects. Associations and aggregations may provide more semantics than just the descriptive attribute name. Both of these relations are bidirectional links. Thus, along with an association there is a reverse association. Similarly, with an aggregation there is a reverse aggregation. An attribute name describes the relation between objects with an association/aggregation type and a slot facet that specifies the reverse relation. A cardinality slot facet can be used to specify the cardinality of the relation.

For example, suppose an employee works for a company, and a company employs the employee. So, creating an Employee class that has an attribute named work_for, which is an association, includes a reverse association, employee_of, linked to a Company class. Also specified is that the cardinality of this work_for relation is many-to-1. Since it is a bidirectional link, in the Company class, an association link employer_of will automatically be created that links to the Employee class. And this employer_of relation has cardinality 1-to-many. So, if a Company class called company_x has John_Doe as an instance of the Employee class, and company_x is the company that John_Doe works for, then John_Doe will be automatically added into the employer_of attribute in company_x that contains a collection of Employee class.

Aggregation represents the part_of relation, and reverse aggregation represents the whole_of relation. Aggregation is a subset of association, except that it is transitive and anti-symmetric.

For example, an airplane has engines, and engines are part of an airplane. To specify this relation, in the Engine class, a power_component_of attribute is added, that is an aggregation, is linked to an Airplane class, and has a reverse aggregation whole_of. Since the link is bidirectional, an aggregation whole_of that links to the Engine class will be created automatically in the Airplane class.

So, modeling relation between objects, using bidirectional association and aggregation links instead of using pointers will produce a more expressive representation.

3.3 Syntax and Semantics

The syntax of the representation language is similar to C++ and can be very simply defined:

class-def	⇒	*classname inheritance* [*constraint*]? *class-body*
inheritance	→	‘(‘ *supername* [*inhspec*]$^?$ ‘)’
supername	→	*name*

inhspec	⇒	'with ALL'
	‖	'::' [*perspective*]
constraint	⇒	'(Constraint:' *decl-clause* ')'
class-body	⇒	'{ [*var-dec*] [*meth-def*] [*pers-dec*]$^{?}$ '}'
var-dec	⇒	*inst-vname* '(' *var-spec* ')' *slot-list*
var-spec	⇒	*classname* [*inhspec*]$^{?}$
	‖	*basic-type*
	‖	'Association'
	‖	'Aggregation'
slot-list	⇒	'(' [*slot-facet*]*')'
slot-facet	⇒	'Cardinality:' *integer*
	‖	'Range:' *range*
	‖	'Minimum:' *integer*
	‖	'Maximum:' *integer*
	‖	'Default:' *number* '|' *string*
	‖	'Prefer:' *number* '|' *string*
	‖	'Type:' *classname*
	‖	'ReverseAssoc:' *string*
	‖	'ReverseAggreg:' *string*
	‖	'Constraint:' *decl-clause*
	‖	'If needed:' *expr-list*
	‖	'Precondition:' *expr-list*
	‖	'Postcondition:' *expr-list*
meth-def	⇒	*methname* '(Method)' [*meth-spec*] '{' *expr-list* '}'
meth-spec	⇒	'(INPUT Cardinality:' *integer* ')'
	‖	'(INPUT:' [*var-dec* ')']*
	‖	'(OUTPUT Cardinality:' *integer* ')'
	‖	'(OUTPUT:' *inh-spec* ')'
	‖	'(Constraint:' *decl-clause* ')'
pers-dec	⇒	'XOR(' [*pers-def*]*')'
	‖	'SET(' [*pers-def*]*')'
	‖	[*pers-def*]*
pers-def	⇒	*persname* '(' *classname* [*pinh-spec*]$^{?}$ ')' [*constraint*]$^{?}$ *pers-body*
pinh-spec	⇒	'with ALL'
	‖	'::' [*perspective*]*
	‖	*persname*
	‖	'Perspective'
pers-body	⇒	'{' [*var-dec*]* [*meth-def*]* [*pers-dec*]$^{?}$ '}'
expr	⇒	use natural language
classname	⇒	*name*
supername	⇒	*name*

methname	$\Rightarrow$	*name*
instvname	$\Rightarrow$	*name*
persname	$\Rightarrow$	*name*
name	$\Rightarrow$	$[a - z][a - z\ 0 - 9]^*$

- The class declaration has class name, superclass (possibly with a list of perspectives), instance variables (attributes), methods (operations), and local perspectives (possibly with SET and XOR constraints).
- The perspective declaration is very similar to the class declaration. It has perspective name, superperspective or superclass (possibly with a list of perspectives), attributes, operations, and possibly local perspectives with SET and XOR constraints.
- The declaration of instance variable has variable name followed by either a basic type, enumerated type, or a class name (possibly with a list of perspectives), and slot facets.
- The method declaration is a subclass of Method class with slot facets on the input and output of the method, followed by an expression-list.
- The basic classes are Object, Perspective, Method, and Reference.
- Object has ObjInit method, which initializes the instance variables of a class.
- Scope of a perspective depends on the place where it is defined. If a perspective is defined within a class or another perspective, then it is only visible within that class/perspective. Otherwise, it is visible globally to everyone.
- Only one level nesting of XOR and SET is allowed to prevent complex nested structure.
- Method can be overloaded through use of input/output slot facets of the method and perspectives.
- Class variables and meta-class operations can be provided at the class level to help reason about the knowledge stored in the class.

3.4 Inheritance

Class inheritance is similar to traditional object-oriented systems, where subclasses inherit default attributes and operations from superclasses. Similarly, perspectives also inherit default attributes and operations from superperspectives. Constraints specified for a class are also inherited to the subclasses.

In addition to inheriting default attributes and operations, a class can also inherit perspectives from its superclasses. Deciding how an object of a class can inherit perspectives from its superclasses was based on an observation of two phenomena, "mutually exclusive relation among defined perspectives" and "inheriting multiple perspectives." These two observations lead to two proposed features for inheriting perspectives: (1) to make the inherited perspectives as part of the defaults (i.e., in addition to default attributes and operations of a class), and (2) to provide some constraint predicates to specify the mutual exclusion and multiple perspectives inheritance among perspectives defined in a class. The following subsections elaborate on these features.

3.4.1 Mutually Exclusive Relation

Often by defining perspectives for a class, what is really meant is that the instanciation/specialization of the class can only have one of the defined perspectives. This exhibits a mutually exclusive relation among the defined perspectives.

Using the previously mentioned queue example, a Queue class is defined with three different perspectives: FIFO Queue P, LIFO Queue P, and Priority Queue P. The Queue class itself has default attributes that define queue and queue elements. But the "enqueue" and "dequeue" operations are defined in each perspective. Clearly, there should be a mutual exclusion relation among these three perspectives since a FIFO queue needs to remove elements from the beginning of the queue, in contrast with a LIFO queue, which needs to remove elements from the end of the queue. So, a specialization of Queue class with FIFO Queue P perspective cannot have LIFO Queue P nor Priority Queue P perspectives at the same time, and vice versa.

On the other hand, to provide better code-reuse, it is best to delay execution of this constraint as long as possible. This means that a subclass of Queue class may inherit all three perspectives (instead of inheriting only one perspective) and may delay the mutual exclusion constraint until the time when one perspective is selected for specialization or instantiation (i.e., to perform the constraint only when necessary). For example, class C2 may become a subclass of Queue class with all perspectives inherited even though there is a mutual exclusivity constraint on these perspectives. Within class C2, Queue class may be refined by including other attributes and operations. The selection of one of the three perspectives can be performed either when the class C2 is instantiated, or when another class C21 is created to be a subclass of class C2 with a subset of perspectives from Queue class. A good example of this kind of delay constraint selection is the Server class defined in the Model-View-Controller[17] (included in the Appendix to this report).

3.4.2 Multiple Perspective Inheritance

A class might want to inherit more than one perspective either implicitly or explicitly. Using the Queue class defined earlier as an illustration, when a class FIFO Queue is created to be a specialization of Queue class with perspective FIFO Queue P, what is really meant is that all the subclasses/instances of FIFO Queue class should have FIFO characteristic. Suppose perspectives P1 and P2 are defined to be local perspectives within FIFO Queue class. If class C1 is created as a subclass of FIFO Queue with perspective P1, then, in addition to perspective P1, class C1 should also inherit perspective FIFO Queue P from FIFO Queue class (originated from Queue class). This is an example of implicit multiple perspective inheritance.

There are also needs for explicit multiple perspective inheritance. A class or an instance of a class may need to have more than one perspective. For instance, in the Traveller-Customer example used in the previous section, in addition to traveller, customer, the class travelling customer would also be desirable. So, instead of having three different classes: traveller, customer, and travelling-customer, it is possible to create a Person class that has Traveller P and Customer P perspectives. Then, a traveller is just a subclass of Person class with Traveller P perspective. A customer is a subclass of Person class with Customer P perspective. And a travelling customer is a subclass of Person class with both Traveller P and Customer P perspectives.

With multiple perspective inheritance, one can create a class from another class with more than one perspective inherited from its superclass. On the other hand, not all the combinations of different perspectives make sense. Thus, constraints must be placed on and among perspectives.

3.4.3 New features for inheriting perspectives

The following two features are proposed for inheriting perspectives: (1) inherited perspectives will be a part of the defaults (this should take care of the implicit multiple perspective inheritance problem), (2) perspective constraint predicates, XOR and SET will be provided with delayed selection (i.e., execution of the constraint is delayed until necessary). XOR is used to specify the mutually exclusive constraint among perspectives, and SET is used to explicitly specify the constraint of multiple perspective inheritance among a set of perspectives.

The following is an example of using XOR. Let C1 be a subclass of Object class, with default attributes and operations defined as D1 (i.e., D1 is the block that defines the attributes and operations). Let P11, P12, and P13 be perspectives defined within C1.

To specify the mutual exclusion relation among P11, P12, and P13 with XOR predicate, do:

```
C1 (Object)
{
        D1
        XOR (
             P11(Perspective) { ... },
             P12(Perspective) { ... },
             P13(Perspective) { ... }
             ) /* end XOR */
}
```

Possible specializations of C1 are:

```
C11 (C1) { ... }
C11 (C1 with P11) { ... }
C11 (C1 with P12) { ... }
C11 (C1 with P13) { ... }
C11 (C1 with ALL) { ... }
```

Note the last one is used for the delayed specialization. On the other hand, an instance of C1 can only have one of the following perspectives (if it has any): P11, P12, or P13.

The following is an example of using SET. Let C1 be a subclass of Object class, with default attributes and operations defined as D1 (i.e., D1 is the block that defines the attributes and operations). Let P11, P12, and P13 be perspectives defined within C1. To specify the constraint for inheriting multiple perspectives among P11, P12, and P13 with SET predicate, we do:

```
C1 (Object) {
  D1
  SET S1 (
  P11(Perspective) { ... } ,
  P12(Perspective) { ... } ,
  P13(Perspective) { ... }
  ) /* end SET */ }
```

This means that an instantiation/specialization of C1 can have the following different combination of perspectives selection: {}, {P11}, {P12}, {P13}, {P11, P12}, {P12, P13}, {P11, P13}, and {P11, P12, P13}. Examples of specialization of C1 are:

```
C11 (C1) { ... }
C11 (C1 with P11) { ... }
C11 (C1 with P12) { ... }
C11 (C1 with P13) { ... }
```

```
C11 (C1 with P11, P12) { ... }
C11 (C1 with P12, P13) { ... }
C11 (C1 with P11, P13) { ... }
C11 (C1 with S1) { ... }
```

XOR and SET can also be used together to specify relations among perspectives. The following is an example of using XOR and SET together.

Let C1 be a subclass of Object class, with default attributes and operations defined as D1 (i.e., D1 is the block that defines the attributes and operations). Let P11, P12, P13, P21, P22, and P23 be perspectives defined within C1.

```
C1 (Object) {
 D1
 XOR (
 P11(Perspective) { ... } ,
 P12(Perspective) { ... } ,
 SET S1 (
        P21(Perspective) { ... } ,
        P22(Perspective) { ... } ,
        P23(Perspective) { ... } ), /* end SET */
 P13(Perspective) { ... }
 ) /* end XOR */ }
```

This means that a specialization of C1 can have the following different combination of perspectives selection:

```
C11 (C1) { ... }
C11 (C1 with P11) { ... }
C11 (C1 with P12) { ... }
C11 (C1 with S1) { ... }
C11 (C1 with P13) { ... }
C11 (C1 with P21) { ... }
C11 (C1 with P22) { ... }
C11 (C1 with P23) { ... }
C11 (C1 with P21, P22) { ... }
C11 (C1 with P22, P23) { ... }
C11 (C1 with P21, P23) { ... }
C11 (C1 with ALL) { ... }
```

Note, that an instance of C1 can have the same combination of perspectives as specialization of C1 displayed above, except for the last one. A limit is also placed on the number of XOR and SET that can be used for each class definition. At most, only one XOR and one SET can be used for each object definition. This avoids the complex nested SET and XOR, which will lead to representations that are hard to understand and manage.

3.5 Defining Class and Perspective Through an Example

This section offers a step-by-step guide to defining classes and perspectives by incrementally building a simple company-employee system. The company-employee system consists of Person Class, Company Class, Employee Class, Manager Class, and Customer Perspective.

Person Class has some default attributes and two perspectives (Traveller and Customer perspectives). Company Class is a subclass of Object, and also has some default attributes and two perspectives (InfoPhoneNum perspective, which contains a list of phone numbers for inquiries, and Customer perspective, which allows a company to act as a customer when dealing with other companies).

Employee Class is a specialization of Person Class with a Traveller perspective (since employee might do company travels). Employee Class itself does not have default attributes, but inherits the default attributes from its superclass according to the inheritance scheme. In addition to Traveller perspective, it also has Company perspective that contains information pertaining to company, PayRoll perspective, which contains information regarding pay check, deduction, and other payroll information, and Employee Purchase perspective, which allows an employee to act as an individual customer of the company he/she works for. An employee's salary should be less than or equal to his/her manager's salary if he/she has a manager.

Manager Class is a subclass of Employee Class, except that it has a collection of Employee Class that a manager would manage. And a manager's salary should be greater than or equal to every employee that he/she manages.

There is also one global perspective, which is Customer perspective. It is a subclass of Perspective, and it provides some common attributes about customers (i.e., billing Address, credit, ...).

The following sections outline the steps starting from defining classes, defining global perspectives (which is similar to defining classes), defining local perspectives, to defining subclasses and two different ways of specializing global perspectives. The convention used in the example is to capitalize the first character of the name of class and perspective. The name of perspective always ends with " P" for easier reading. Perspective and Object are the basic types, so in String, Integer, Address, ... Class, perspective, and attribute are parenthesized immediately followed the declaration. Comments are preceeded with "/*" and ended with "*/" similar to programming conventions in C.

3.5.1 Defining Classes

A class/perspective definition is composed of the following parts: its name, superclass, default attributes and operations, locally defined perspectives, and inclusion/-specialization of global perspectives. The following will focus on specifying the default attributes and operations. For example, a Person class, with four default attributes: firstName, lastName, age, and homeAddress, can be defined as follows:

```
Person(Object){
        firstName(String)
        lastName(String)
        age(Integer)
        homeAddress(Address)
}
```

Similarly, a Company class, with three default attributes: name, stAddress, and divNum, can be defined as follows:

```
Company(Object){
        name (String)
        stAddress (Address)
        divNum (String)
}
```

3.5.2 Defining Global Perspectives

Customer P is a globally defined perspective, and has two attributes: billingAddress, and credit. It can be included in another class (i.e., a perspective of a certain class) or as a superperspective of another perspective.

```
Customer P(Perspective){
        billingAddress(Address)
        credit (XOR credit card, cash, check, money order)
}
```

The following example demonstrates the inclusion of the Customer perspective in the class definition without specializing:

```
Person(Object){
        firstName(String)
        lastName(String)
        age(Integer)
        homeAddress(Address)
        Customer P{}
}
```

3.5.3 Defining Local Perspectives

Perspectives can also be defined locally within a class as part of its components. A perspective can be either a subperspective of Perspective class, or a subperspective of a global perspective. The latter is detailed in the section "Specializing Global Perspectives: Within Subperspectives." (p 31)

Using the Person class and the Company class defined previously as the basis, a Person class with a local Traveller P perspective can be defined as:

```
Person(Object){
        firstName(String)
        lastName(String)
        age(Integer)
        homeAddress(Address)

  SET(
        Traveller  P(Perspective)
        {
              age (XOR Infant Child Adult)

              (IF  NEEDED: calculated from the age in person)
              preferredAirport (String)
              (IF  NEEDED: Calculate from the state in the homeAddress)
        },
        Customer  P{}
  )
}
```

Within the locally defined Traveller P perspective, there are two attributes, age and preferredAirport. The age attribute is an enumerated type and is calculated from the age default attribute in Person class. The preferredAirport attribute is a string type and is obtained from the state attribute in the homeAddress default attribute in Person class. These attributes are only accessible within Person class, and are not defined outside of Person class context.

Company class with a locally defined InfoPhoneNum P perspective, which contains phone numbers for various information inquiries, can be defined as follows:

```
Company(Object){
        name (String)
        stAddress (Address)
        divNum (String)
        InfoPhoneNum  P(Perspective){
              personnel(PhoneNum)
              receptionist(PhoneNum)
              press(PhoneNum)
        }
}
```

3.5.4 Defining SubClasses

Once a class is defined, it can be further refined/specialized into subclasses. There are three ways of declaring a subclass when specifying the superclass, depending on the number of perspectives that the subclass inherits from its superclass.

Let class B be a subclass of class A. Subclass B can inherit either zero perspectives from A or one or more perspectives from A. Let P1, P2, and P3 be perspectives defined in class A. Declaration of "B(A) .... " will make class B a subclass of class A without inheriting any perspectives, thus only the default attributes and operations of A are inherited by B. Declaration of "B(A with P1)" or "B(A with P1, P2)" will make class B be a subclass of class A with perspective P1 (or P1 and P2). Thus, in addition to the default attributes and operations of A, class B also inherits perspective P1 (or P1 and P2). To inherit all the perspective of A, class B can be declared as "B(A with ALL)". More detail on the subject of inheritance is included in the section on inheritance (p 22).

For example, Employee is a person who also does company travelling. Thus, it is a subclass of Person class with Traveller perspective. And in addition to the inherited Traveller perspective, it also has a local perspective PayRoll that contains all the payroll information about the employee. The following is the declaration of Employee class.

```
Employee(Person::Traveller  P){
        PayRoll  P(Perspective){
              deduction(Money)
              tax(Money)
              withhold(Money)
              net(Money)
        }
}
```

Notice that here, the default attributes and operation of Employee class are the ones that it inherited from its superclass. Employee class itself does not define its own default attributes and operations.

3.5.5 Specializing Global Perspectives: During Inclusion

Attributes in the globally defined perspectives can be further refined, specialized, and overwritten either by the subperspective (defined globally or locally within the class definition) or during the inclusion of global perspectives in the class definition.

This subsection focuses on the specialization of global perspectives during their inclusion in the class definition. An example of attribute refinement at the inclusion time can be found in the Company class.

Based on the Company class defined earlier, a company also purchases things from other companies. Thus, it should also have Customer perspective. But, the globally defined Customer perspective does not specify how to find the proper billingAddress. So, the billingAddress in the Company class must be refined so that the billingAddress will be calculated from the stAddress attribute of the Company class. The following example demonstrates the idea:

```
Company(Object){
        name (String)
        stAddress (Address)
        divNum (String)
  SET (
        InfoPhoneNum  P(Perspective){
        personnel(PhoneNum)
        receptionist(PhoneNum)
        press(PhoneNum)
  },
        Customer  P{
             billingAddress(IF  NEEDED: get from the stAddress in company)
        }
  )
}
```

Note that the difference between the usage of global perspective in the Company class and the Person class is that, in the Company class, the billingAddress attribute of Customer P is further specialized. It specifies how to obtain the value of billing Address (from the stAddress attribute in the Company class) via IF NEEDED slot.

3.5.6 Specializing Global Perspectives: Within Sub-Perspectives

As mentioned earlier, attributes can also be further refined in a subperspective. Subperspectives can be defined globally or locally within the class definition. Since many companies provide employee purchasing programs whereby employees can purchase company products at special discounts, an Employee purchase perspective should be added to the Employee class. When viewed in the Employee purchase perspective, an employee is like a customer to the company, except that the preferred billingAddress is the homeAddress of the employee and the credit (the way to pay for the merchandise) can be deducted from payroll in addition to the usual payment options of a normal customer. Thus, Employee purchase perspective is a sub-perspective of Customer perspective with a refinement of the billingAddress and the credit attributes.

Also, since an employee is working in a company, the information about a company should be included in the employee class. And whenever the employee is doing any purchase for the company, the bill should go to the employee's company instead of to the employee. So a local perspective Company P, which is a subperspective of Company class with Customer perspective, needs to be defined for the Employee class. In addition to the defaults that it inherits from Company class and Customer perspective, Company P perspective also defines some other attributes relating to Employee, i.e., employeeID, salary, title, department name, telephone number, and the person who manages him/her. The constraint for the salary attribute is that an employee's salary should be less than or equal to his/her manager's salary. The definition for this Employee class is given in the following:

```
Employee(Person::Traveller P){
      SET(
           Company P(Company::Customer P){
                employeeID(String)
                salary(Money)
                (Constraint: less than equal to(salary(self), salary(manager)))
                title(TitleString)
                depName(DepartmentString)
                telPhone(String)
                manager(Manager)
           },

           PayRoll P(Perspective){
                deduction(Money)
                tax(Money)
                withhold(Money)
                net(Money)
           },
            Employee purchase P(Customer P){
                billingAddress(PerferredValue: get from the homeAddress)
                credit(ADD payroll)
            }
        )
}
```

Manager Class is a subclass of the Employee class with one additional attribute that contains a collection of employees that a manager manages. And the salary of a manager should be greater than or equal to the employees that he/she manages. It can be defined as follows:

```
Manager(Employee with ALL)
        (Constraint: member (each, manage),
            greater than equal to (salary(self),salary(each)))
{
        manage(Collection of Employee)
}
```

Note that the Employee class by itself has three different perspectives: Company P, PayRoll P, Employee purchase P (i.e., company employee, payrolled employee, and purchase employee.) Furthermore, a perspective can be a specialization of a certain class (as the Company P perspective), specialization of a global perspective (as the Employee purchase P perspective), or just a locally defined one (as the PayRoll P perspective). Company P perspective is a specialization of Company class with Customer P perspective, so it inherits the default attributes of Company class as well as the attributes of Customer P that is refined in the Company class. It also defines some more attributes for itself, i.e., employeeID, salary, etc. PayRoll P perspective is a locally defined perspective with four default attributes. Employee purchase P perspective is a locally defined subperspective of Customer P perspective. The credit attribute inherits the enumerated type information from the Customer P perspective with payroll appended to the inherited enumerated type through use of ADD.

However, this is not a very good way to define the Employee Class because information about a company is redundantly stored in every instance of the Employee Class. A similar problem also exists for the Manager Class. A better way to define the Employee Class is to define the local perspective Company P as a subperspective of the Perspective class. In addition to the attributes defined in the former Company P perspective of the Employee Class, i.e., employeeID, salary, etc; this new Company P perspective also defines a work for association that is linked to a Company class and has a reverse association employer of. Similarly, for the same reason, a managed by association can be defined for the Employee Class and the Manager Class. The modified Employee Class and Manager Class are as follows:

```
Employee(Person::Traveller_P){
        SET(
            Company_P(Perspective)
            {
                employeeID(String)
                salary(Money)
                (Constraint: less_than_equal_to(salary(self), salary(managed_* *by)))
                title(TitleString)
                depName(DepartmentString)
                telPhone(String)
                work_for
                    (Association)
                    (ReverseAssociation: employer_of)
                    (Type: Company)
                    (Cardinality: Many to 1)
                managed_by
                    (Association)
                    (ReverseAssociation: manage)
                    (Type: Manager)
                    (Cardinality: Many to 1)
                billingAddress(get from the billingAddress in work_for)
                credit(get from the credit in work_for)
        },

        PayRoll_P(Perspective)
        {
            deduction(Money)
            tax(Money)
            withhold(Money)
            net(Money)
        },

        Employee_purchase_P(Customer_P)
        {
            billingAddress(PerferredValue: get from the homeAddress)
            credit(ADD payroll)
        }
        )
}

Manager(Employee with ALL)
        (Constraint: member(each, manage),
            greater_than_equal_to(salary(self), salary(each)))
{}
```

The resultant Company-Employee system is as follows:

```
Customer  P(Perspective){
        billingAddress(Address)
        credit (XOR credit  card, cash, check, money  order)
}

Person(Object){
        firstName(String)
        lastName(String)
        age(Integer)
        homeAddress(Address)

        SET (
             Traveller  P(Perspective){
                  age (XOR Infant Child Adult)
                  (IF  NEEDED: calculated from the age in person)
                  preferredAirport (String)
                  (IF  NEEDED: Calculate from the state in the homeAddr* *ess)
             },

             Customer  P{}
        )
}

Company(Object){
        name (String)
        stAddress (Address)
        divNum (String)

        SET (
             InfoPhoneNum  P(Perspective){
                  personnel(PhoneNum)
                  receptionist(PhoneNum)
                  press(PhoneNum)
             },

             Customer  Pf
                  billingAddress(IF  NEEDED: get from the stAddress in company)
             }
        )
}

Employee(Person::Traveller  P){
        SET (
                  Company  P(Perspective){
                       employeeID(String)
                       salary(Money)
                       (Constraint:
                              less  than  equal  to(salary(self), salary(managed  by)))
                        title(TitleString)
                        depName(DepartmentString)
                        telPhone(String)
                        work  for(Association)
                              (ReverseAssociation: employer  of)
                              (Type: Company)
                              (Cardinality: Many to 1)
                        managed  by(Association)
                              (ReverseAssociation: manage)
                              (Type: Manager)
                              (Cardinality: Many to 1)
                  billingAddress(get from the billingAddress in work  for)
                  credit(get from the credit in work  for)
            },

            PayRoll  P(Perspective){
                  deduction(Money)
                  tax(Money)
                  withhold(Money)
                  net(Money)
            },

            Employee  purchase  P(Customer  P){
                  billingAddress(PerferredValue: get from the homeAddress)
                  credit(ADD payroll)
            }
      )
}
Manager(Employee with ALL)
      (Constraint: member(each, manage),
            greater  than  equal  to(salary(self), salary(each)))
{}
```

4 Conclusions

4.1 Summary

This study has found that the traditional object-oriented system is not powerful enough to represent knowledge about an object, and the information stored in an object is not enough to reason about the object. A new representation is proposed to provide enough semantic information stored in the object structure that will help us to reason about it.

The proposed augmented features are:

- attaching slot facets to attributes to provide more semantic information
- using perspectives to group together the relevent information about an object
- adding a bidirectional link for the superclass/subclass relation
- adding a bidirectional link for the class/perspective relation
- adding constraint, invariants, security, and integrity checking either on the operation/attribute or on the class itself
- adding bidirectional association and aggregation links to describe the relation between objects.

For inheritance, in addition to the inheritance of a class and inheritance of a perspective, multiple perspective inheritance was allowed and the capability of specifying constraints among perspectives provided through the use of XOR and SET predicates. Constraint, invariants, security, and integrity checking specified on the operation/attribute or on the class are also inherited to the subclasses.

Advantages of using perspectives are :

- localizing the relevent information (attributes and operations)
- providing more semantic information to attributes and operations—providing overloading of operations and attributes
- providing information at different levels of details through perspectives and refinement.

Advantages of using association and aggregation are:

- avoiding storing redundant information
- modeling the relation between objects using links instead of pointers.

4.2 Future Research

This study has proposed an extension of the object-oriented formalism that will provide greater support for knowledge representation and knowledge-based reasoning. With regards to this language formalism, a number of interesting questions still need to be considered:

- whether multiple perspective inheritance is needed
- whether the ability to specify perspectives within perspectives is needed
- what the language implementation for this representation is
- how to handle naming conflicts.

4.2.1 Multiple Perspective Inheritance

In the proposed knowledge representation, "SET", "XOR", and the ability to make inherited perspectives a part of the defaults provide some flavors of multiple perspective inheritance. These are all alternatives to multiple inheritance. At this point whether to provide multiple inheritance has not been determined. This issue will be studied more in the simulation modeling domain, and addressed later.

4.2.2 Defining Perspectives within Perspectives

Being able to define perspectives within another perspective is a good feature, but one that does not add much power to the representation. Also, the study did not encounter any examples that require this feature. For now, one level perspective definition will suffice. In the future, if needed, the ability to define perspectives within perspectives can be added into the representation. Note that the current representation does not prevent defining perspectives within perspectives.

4.2.3 The Language Implementation

The language implementation for the representation will be a combination of imperative and declarative languages, similar to the IMPORT/DOME language designed at USACERL [10]. The imperative part is used to define classes, perspectives, attributes, and operations. The declarative part is used to describe the constraints, invariants, security, and integrity checking for the attributes, operations, and objects as well as the slot facets for the attributes, and operations.

Based on observation, imperative languages are known to be procedural, structural, and modular. So, it is more suitable to use imperative language to define the classes, perspectives, attributes, and operations that require more structure and modularity. On the other hand, declarative languages are known to be very expressive; the theorem prover is powerful enough to deduce information from a large collection of facts. Also, declarative languages allow one to assert facts and to define predicates. Since one cannot foresee the type of constraints that a class, an attribute, or an operation may have, nor is it desirable to limit the type of constraints that one can have, it is a good idea to use the expressive declarative language to specify the constraints and slot facets.

Constraints of an object defined by the declarative languages can be checked by the theorem prover, when a request to access the object is made. If constraints are violated, then the operation requested will not be executed. Also an object may want to query another object's constraints or an attribute's slot facets. This can also be done by the theorem prover. Since the reference to an object can be passed into the theorm prover, the symbol table is available: reference to objects can be found and the theorem prover can access information of other objects.

The compiler for the implementation language can be a two-phase compiler using an object-oriented database. The first phase parses the source file, performs static semantic checking, generates an intermediate form, and stores it in an object-oriented database. The programming tools can be built to use the intermediate form for debugging and running the interpreter to allow fast prototype. The second phase converts the intermediate form into the desired programming language such as C++, an approach currently used in the IMPORT/DOME language.

4.2.4 Naming Conflict

Because of the SET constraint for perspectives, naming conflicts might arise. A number of possible strategies exist for dealing with such conflicts; perhaps the simplest is to use the ordering sequence of the class/perspective being declared in a class definition.

4.3 Summary

In object-oriented approaches, inheritance and polymorphism provide mechanisms for sharing implementations, with the ultimate goal of producing reusable modules, sometimes referred to as "software ICs." In frame-based approaches, inheritance provides a mechanism for prototypical reasoning, where assumptions are inherited and refined as specific information becomes available.

Although the motivation behind the concepts is quite different, a number of striking similarities exist between the two approaches, mostly at the philosophical and methodological levels:

- Both decompose a system into a collection of objects and relationships.
- Both encapsulate or bundle knowledge into groups of attributes.
- Both use a similar structure (named slot-value pairs).
- Both incorporate a notion of inheritance.
- Both include a notion of individual active entities.

The central issues with regards to integrating the two approaches involves some fundamental issues: whether reusability of software maps into prototypical knowledge; or whether software reuse can be realized in the presence of prototypical inferencing strategies. These issues require more research, and will probably only yield answers when these ideas are implemented in an experimental environment.

Bibliography

[1] Daniel Bobrow and Terry Winograd. "An Overview of KRL, a Knowledge Representation Language." In R. Brachman and H. Levesque, editors, *Readings in Knowledge Representation,* chapter 13, pages 245-262. Morgan Kaufmann Publishers, Inc, Los Altos, CA, 1985.

[2] D.G. Bobrow and T. Winograd. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Science*, 1(1), January 1977.

[3] W.C. Dietrich, L.R. Nackman, and F. Gracer. "Saving a Legacy With Objects." In *OOPSLA*, 1989.

[4] C.L. Dym and R.E. Levitt. *Knowledge-Based Systems in Engineering*. McGraw-Hill, 1991.

[5] R. Fikes and T. Kehler. "The Role of Frame-Based Representation in Reasoning." *Communications of the ACM*, September 1985.

[6] Patrick J. Hayes. "The Logic of Frames." In R. Brachman and H. Levesque, editors, *Readings in Knowledge Representation*, chapter 14, pages 287-295. Morgan Kaufmann Publishers, Inc, Los Altos, CA, 1985.

[7] Charles Herring and R. Alan Whitehurst. *Concept Design for an Object-Oriented Database Capability in MODSIM*. Letter report to U.S. Army TRADOC Analysis Command, U.S. Army Construction Engineering Research Laboratory, 1991.

[8] C. Herring and R.A. Whitehurst. "Adding Peristence to an Object-Oriented Simulation Language." In *Society for Computer Simulation Multiconference on Object-Oriented Simulation*, San Diego, CA, 1991. Simulation Councils, Inc.

[9] Charles Herring. "Army Model Hierarchy = Technology*5 + Architecture + Methodology." In *U.S Army Operations Reasearch Symposium, Proceedings*, November 1993.

[10] Charles Herring J. Teo, V. Karamcheti, and R Alan Whitehurst. *Definition and Implementation of the Integrated Modular Persisent Object Representation Translator (IMPORT)*. Technical Report P-93/11/ADA273355, U.S. Army Construction Engineering Research Laboratories, Champaign, IL, September 1993.

[11] Charles Herring, Biju Kalathil, and Joseph Teo. *Research in Persistent Simulation: Development of the Persistent ModSim Object-Oriented Programming Language*. Technical Report P-93/07/ADA268568. U.S. Army Construction Engineering Research Laboratories, July 1993.

[12] Charles Herring, Jeffrey Wallace, and R. Alan Whitehurst. *Application of Object-Oriented Programming to Combat Modelling and Simulation.* Technical Report P-91/46/ADA242673. U.S. Army Construction Engineering Research Laboratories, September 1991.

[13] Charles Herring, Jeffrey Wallace, R. Alan Whitehurst, and David Adams. "Design of an Engineer Functional Area Model Using Next Generation Software Tools and Methodology." In *U.S Army Operations Reasearch Symposium, Proceedings*, November 1991.

[14] Charles Herring and R. Alan Whitehurst. *Application Profile: Requirements for Persistent Simulation*, Position Paper submitted to the *DARPA Open Object-Oriented Database Workshop.* Texas Instruments, March 1991.

[15] Ralph Johnson. *Cs499 Object-Oriented Programming.* Class Lecture, August 1990.

[16] Biju J. Kalathil and Charles Herring. "System Support for Assembling Compatible Configurations in an Integrated Programming Environment." Submitted to *Software Configuration Management 1993*, May 1993.

[17] G.E. Krasner and S.T. Pope. "A Cookbook for Using the Model-Viewcontroller User Interface Paradigm in Smalltalk-80." *Journal of ObjectOriented Programming*, pages 26-49, August 1988.

[18] Bertrand Meyer. "Object-Oriented Programming." In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Applications and Experience*, chapter 1, pages 1-33. Addison-Wesley, Reading, MA, 1989.

[19] Marvin Minsky. "A Framework for Knowledge Representation." In J. Haugeland, editor, *Mind Design*, pages 95-128. MIT Press, Cambridge, MA, 1981.

[20] Marvin Minsky. *The Society of the Mind.* Simon and Schuster, New York, 1986.

[21] Oscar Nierstrasz. "A Survey of Object-Oriented Concepts." In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 1, pages 3-22. Addison-Wesley, New York, 1989. ACM Press Frontier Series.

[22] Director of Defense Research and Engineering. *Defense Science and Technology Plan*. July 1992.

[23] N.W. Paton and O. Diaz. "Object-Oriented Database and Frame-Based Systems: Comparison." *Software and Information Technical Journal*, 1987.

[24] E. Rich and K. Knight. *Artificial Intelligence.* McGraw-Hill, second edition, 1991.

[25] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[26] R. Alan Whitehurst. "Simulation Utilizing an Interpretive Object-Oriented Rule-Based Approach." In Raimund K. Ege, editor, *Object-Oriented Simulation*. Computer Simulation Society, January 1991.

[27] Patrick Henery Winston. *Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, MA, second edition, 1984.

[28] S.J. Young and C. Proctor. "An Experimental Frame Language Based on Abstract Data Types." *The Computer Journal*, 29(4), 1986.

Appendix: Example Model View Controller

The following Model View Controller system uses the new representation. The Model View Controller system is a network queue simulation system that has three parts: model, view, and controller. The model uses servers, queues, and a token to model the queueing network. A model can have a number of servers and each server has one queue. A token is passed around in the queueing network; whoever gets the token gets service time. View provides ways to look at the status of the current queueing network. It can display this information in many different formats. Controller mediates the interaction between a model and a view. Whenever a view wants to display the status, it asks the controller for the data. Controller looks into the status of the model, calculates the result, and sends it to the view for display. Each server has a different distribution that simulates the rate of service requests coming into the queue. Thus it is a subclass of a random number generator object. Queue can be either FIFO, LIFO, or priority-based. Queue is a linked list of queue elements that provides add and remove operations to add and remove elements from the queue.

```
/* globally defined perspective */

Queue  opt  P (Perspective) {
Add (Method) {};
Remove (Method) {};
}

Queue  Elm (Object) {
item (Reference);
next (Reference);
setItem (Method)
(INPUT  Cardinality: 1)
(INPUT: ref (Reference))
(OUTPUT  Cardinality: 0) {
item = ref;
}

setNext (Method)
        (INPUT  Cardinality: 1)
        (INPUT: ref (Reference))
        (OUTPUT  Cardinality: 0) {
next = ref;
}
```

```
getNext (Method)
        (INPUT  Cardinality: 0)
        (OUTPUT  TYPE: (Reference)) {
 return next;
}

} /* end of Queue  Elm */

Queue (Object) {
 /* defaults for Queue */
numberIn (Integer);
first (Reference); /* pointer */
last (Reference); /* pointer */

QueueInit (Method)
        (INPUT  CARDINALITY: 0)
        (OUTPUT  TYPE: Queue) {
ObjInit (self);
  numberIn = 0;
  first = NIL;
  last = NIL;
 return self;
 }

XOR (
FIFO  Queue  P (Queue  opt  P) {
Add (Method)
(INPUT: elm (Reference))
(OUTPUT  Cardinality: 0)
(LOCAL  VAR: new  elm (Queue  elm)) {

new  elm = send NewObj(Queue  Elm);
send new  elm setItem (elm);
send new  elm setNext (NIL);
numberIn = numberIn + 1;
If (numberIn > 1) {
send *last setNext
(send new  elm GetReference);
last = send *last getNext;
} else {
first = send new  elm GetReference;
last = first;
}
 }

Remove (Method)
        (INPUT  Cardinality: 0)
        (OUTPUT: (Reference))
        (LOCAL  VAR: rm  elm (Queue  elm)) {
```

```
If (numberIn > 0) {
rm  elm = send first Ref2Obj;
first = send rm  elm getNext;
send rm  elm setNext (NIL);
numberIn = numberIn 1;
return (send rm  elm GetReference);
 } else {
  return NIL:
 }
 }
 }, /* end of FIFO  Queue  P */

LIFO  Queue  P (Queue  opt  P) {
Add (Method)
        (INPUT: elm (Reference))
        (OUTPUT  Cardinality: 0)
        (LOCAL  VAR: new  elm (Queue  elm)) {

new  elm = send NewObj(Queue  Elm);
send new  elm setItem (elm);
send new  elm setNext (NIL);
numberIn = numberIn + 1;
If (numberIn > 1) {
send new  elm setNext (last);
last = (send new  elm GetReference);
} else {
send new  elm setNext (NIL);
last = (send new  elm GetReference);
first = last;

}
}

Remove (Method)
        (INPUT  Cardinality: 0)
        (OUTPUT: (Reference))
        (LOCAL  VAR: rm  elm (Queue  elm)) {

If (numberIn > 0) {
rm  elm = send last Ref2Obj;
last = send rm  elm getNext;
send rm  elm setNext (NIL);
numberIn = numberIn 1;
return (send rm  elm GetReference);
} else {
return NIL:
}
}
}, /* end of LIFO  Queue  P */

 Priority  Queue  P (Queue  opt  P) {
```

```
Add (Method)
        (INPUT: elm (Reference))
        (OUTPUT Cardinality: 0)
        (LOCAL VAR: new elm (Queue elm)) {

new elm = send NewObj(Queue Elm);
send new elm setItem (elm);
send new elm setNext (NIL);
numberIn = numberIn + 1;
If (numberIn > 1) {
/*** Insert it to the proper place ***/
} else {
send new elm setNext (NIL);
last = (send new elm GetReference);
first = last;
}
}

Remove (Method)
        (INPUT Cardinality: 0)
        (OUTPUT: (Reference))
        (LOCAL VAR: rm elm (Queue elm)) {
If (numberIn > 0) {
rm elm = send last Ref2Obj;
last = send rm elm getNext;
send rm elm setNext (NIL);
numberIn = numberIn 1;
return (send rm elm GetReference);
} else {
return NIL:
}
}
} /* end of Priority Queue P */
) /* end of XOR */
} /* end of Queue */

RandomNum (Object) {
str1, stsr2, str3, str4 (Integer);
p1, p2, p3 (Real);
ip1,ip2 (Integer);
dtype (DistType);
rvtype (RVType);

InitRVObj(Method)
        (INPUT Cardinality: 3)
        (INPUT: lo,hi (Real); stream (Integer))
        (OUTPUT Cardinality: 0) { ... }
        XOR (
UniformReal P(Perspective) {
InitRVObj(Method)
        (INPUT Cardinality: 3)
        (INPUT: lo,hi (Real); stream (Integer))
        (OUTPUT  Cardinality: 0)
{... InitRVObj in RamdomNum ...}

GenerateContinuousRV(Method)
        (INPUT  Cardinality: 0)
        (OUTPUT: (Real)) { ... }
},
 UniformInt  P(Perspective) {
InitRVObj(Method)
(INPUT  Cardinality: 3)
(INPUT: low, high, str (Integer))
(OUTPUT  Cardinality: 0)
{ ... InitRVObj in RamdomNum ... }

GenerateDiscreteRV(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Integer)) { ... }
},

Exponential  P(Perspective) {
InitRVObj(Method)
(INPUT  Cardinality: 2)
(INPUT: mu (Real); stream (Integer))
(OUTPUT  Cardinality: 0)
{ ... InitRVObj in RamdomNum ... }

GenerateContinuousRV(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Real)) { ... }
},

 Normal  P(Perspective) {
InitRVObj(Method)
(INPUT  Cardinality: 4)
(INPUT: mu,sigma (Real); stream1, stream2 (Integer))
(OUTPUT  Cardinality: 0)
{ ... InitRVObj in RamdomNum ... }

 GenerateContinuousRV(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Real)) { ... }
},

 Gamma  P(Perspective) f{
InitRVObj(Method)
(INPUT  Cardinality: 4)
(INPUT: alph,beta (Real); stream1, stream2 (Integer))
(OUTPUT  Cardinality: 0)
{ ... InitRVObj in RamdomNum ... }
```

```
GenerateContinuousRV(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Real)) { ... }
},

Beta  P(Perspective) {
InitRVObj(Method)
(INPUT  Cardinality: 6)
(INPUT: k1, k2 (Real); str1,str2, str3,str4(Integer))
(OUTPUT  Cardinality: 0)
{ ... InitRVObj in RamdomNum ... }

GenerateContinuousRV(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Real)) { ... }
},

 Triangular  P(Perspective) {
InitRVObj(Method)
(INPUT  Cardinality: 4)
(INPUT: a,b,c (Real); stream(Integer))
(OUTPUT  Cardinality: 0)
{ ... InitRVObj in RamdomNum ... }

GenerateContinuousRV(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Real)) { ... }
},

Weibull  P(Perspective) {
InitRVObj(Method)
(INPUT  Cardinality: 3)
(INPUT: a,b (Real); stream(Integer))
OUTPUT  Cardinality: 0)
{ ... InitRVObj in RamdomNum ... }

GenerateContinuousRV(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Real)) { ... }
}
        ) /* end of XOR */
}

}

Server (RandomNum with ALL) {
id (Integer);
status (Real);
dependents (List);
```

```
ServerObjInit(Method)
(INPUT  Cardinality: 0)
(OUTPUT  Cardinality: 0) { ... }

IsBusy(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Boolean)) { ... }

MakeIdle (Method)
(INPUT  Cardinality: 0)
(OUTPUT  Cardinality: 0) { ... }

MakeBusy (Method)
(INPUT  Cardinality: 0)
(OUTPUT  Cardinality: 0) { ... }

GenerateServiceTime(Method)
(INPUT  Cardinality: 0)
(OUTPUT: (Real)) { ... }

SetQueue(Method)
(INPUT  Cardinality: 1)
(INPUT: q (Queue))
(OUTPUT  Cardinality: 0) { ... }

ProcessToken(Method)
(INPUT  Cardinality: 1)
(INPUT: part (Reference))
(OUTPUT  Cardinality: 0) { ... }

XOR (
FIFO  P (Perspective) {
queue (Queue with FIFO  Queue  P);
},

LIFO  P (Perspective) {
 queue (Queue with LIFO  Queue  P);
},

Prio  P (Perspective) {
queue (Queue with Priority  Queue  P);
 }
); /* end of XOR */
}

Token (Object) {
id (Integer);
arrivalTime (Real);
dueTime (Real);
systemArrivalTime (Real);
```

```
setId (Method)
(INPUT  Cardinality: 1)
(INPUT: id (Integer))
(OUTPUT  Cardinality: 0) { ... } ;

setArrivialTime (Method)
(INPUT  Cardinality: 1)
(INPUT: at (Real))
(OUTPUT  Cardinality: 0) { ... } ;

setDueTime (Method)
(INPUT  Cardinality: 1)
(INPUT: dt (Real))
(OUTPUT  Cardinality: 0) { ... } ;

setSystemArrivialTime (Method)
(INPUT  Cardinality: 1)
(INPUT: at (Real))
(OUTPUT  Cardinality: 0) { ... } ;
}

View (Object with Display  P) {
...
}

Server  ViewControllerObj (Object) {
Model (Reference);
View (Reference);
AveServerUtilization (Real);

Update (Method)
(INPUT  Cardinality: 0)
(OUTPUT  Cardinality: 0) { ... } ;
}
```

An instance of Server object will be:

```
Server  1 (Server with UniformReal  P and FIFO  P)
```

Note that there are two perspectives: UniformReal P and FIFO P, which need to be specified in order to make an instance of Server. UniformReal P perspective is used to specialize Server 1 to be a RandomNum object with UniformReal P perspective (since Server object is a subclass of RandomNum with ALL perspectives inherited). And FIFO P perspective is used to specialize Server 1 to be a server using a FIFO queue (since FIFO P is defined in the Server object to distinguish the different type of queue).

DISTRIBUTION

Chief of Engineers
ATTN: CEHEC-IM-LH (2)
ATTN: CEHEC-IM-LP (2)
ATTN: CERD-L
ATTN: CECC-R

Defense Technical Information Center 22304
ATTN: DTIC-FAB (2)

8
12/94